Tree Bitmap : Hardware/Software IP Lookups with Incremental Updates

W. Eatherton, Z. Dittia, G. Varghese

Abstract

IP address lookup is challenging for high performance routers because it requires a longest matching prefix at speeds of up to 10 Gbps (OC-192). Existing solutions have poor update times or require large amounts of expensive high speed memory, and do not take into account the flexibility of ASICs or the structure of modern high speed memory technologies such as SDRAM and RAMBUS. In this paper, we present a family of IP lookup schemes using a data structure that compactly encodes large prefix tables. For example, one of our reference implementations requires only 110 Kbytes of memory for the 41,000 entry Mae East database). The schemes can be instantiated to require a maximum of 4-7 memory references which, together with a small amount of pipelining, allows wire speed forwarding at OC-192 (10 Gbps) rates. We also present a series of optimizations to the core algorithm that allows the memory access width of the algorithm to reduced at the cost of memory references or allocated memory.

1 Introduction

Recently IP destination address lookups have received a great deal of attention [3][4][12][18][20][21][22]. Because of increased traffic in the Internet, faster links are being deployed; 2.48 Gbit/sec backbone links are being deployed, and the time is not far off when 10 Gigabit/sec (OC-192c) links will be deployed. Faster links in turn require faster forwarding by routers. Since a large number of Internet packets are minimum sized packets (e.g., TCP acks), a number of vendors advertise wire speed forwarding, which is the ability to forward minimum size TCP/IP packets at line rates. Wire speed forwarding for say OC-192c rates requires 24 million forwarding decisions (and hence IP lookups) per second.

While there are other forwarding tasks, the basic tasks such as TTL updates and checksums can easily be done in hardware; scheduling can be handled efficiently using either FIFO, RED, or weighted round robin schemes; finally, filter processing for up to 1000 filters can be handled by a ternary CAM. Unfortunately, backbone routers today can easily have 50,000 prefixes (taking growth into account, it is reasonable for a backbone router today to aim for several hundred thousand prefixes) for which brute force solutions like CAMs are too expensive. For these reasons, fast, cheap algorithmic solutions to the IP lookup problem with deterministic performance guarantees are of great interest to router vendors.

IP lookups require a longest matching prefix computation at wire speeds. The current version (IPv4) uses 32 bit destination addresses; each Internet router can have say 50,000 prefixes, each of which we will denote by a bit string (e.g., 01*) of up to 32 bits followed by a ‘‘*’’. Each prefix entry consists of a prefix and a next hop value. For example, suppose the database consists of only two prefix entries (01* --> L1; 0100* --> L2) If the router receives a packet with destination address that starts with 01000, the address matches both the first prefix (01*) and the second prefix (0100*). Since the second prefix is the longest match, the packet should be sent to next hop L2. On the other hand, a packet with destination address that starts with 01010 should be sent to next hop L1. The next hop information will typically specify an output port on the router and possibly a Data Link address.

What are the requirements of an ideal IP lookup scheme? First, it should require only a few memory

accesses in the worst case to allow wire speed forwarding. Second, it should require as small an amount of expensive high speed memory as possible. In particular, if the IP lookup algorithm is implemented as a single chip solution, the entire data structure (including overhead) must fit into the maximum amount of on-chip memory.

Besides determinism in terms of lookup speed and storage, many vendors would also like determinism in terms of update times. Many existing IP lookup schemes (e.g., [3],[4],[12], [18],[22]) are tailored towards fast lookup speeds but have poor worst case update performance. The first reason for fast update times is the presence of routing instabilities [11], which can result in updates every millisecond. Slow update performance can result in dropped updates and further instability. For single chip solutions, a better reason is simplicity. If the update process is simple and takes a bounded amount of time, the chip (as opposed to the software) can do the updates in hardware. This makes the software much simpler and makes the chip more attractive to customers.

Although the preceding discussion was slanted towards hardware, we would also like a core lookup algorithm that can be tuned to work in differing environments including software, single chip implementations, and single chip with supporting memory. Even when using off-chip memory, there are several interesting memory technologies. Thus a last important goal of a useful IP lookup scheme is tunability across a wide range of architectures and memories. In Section 2, we present a new model for memory architectures that abstracts the essential features of many current memory configurations. This model can be used to tune any IP lookup algorithm by suggesting optimum access sizes, which in turn affects the way data structures are laid out.

In this paper we provide a systems viewpoint of IP lookups, with heavy emphasis on updates. We provide an analysis of current and future memory technologies for IP lookups in hardware. We present a new algorithm, named Tree Bitmap, that we show has good update qualities, fast lookup times, low storage needs, and is amenable to both software and hardware implementations. We present a family of optimizations to Tree Bitmap that combined with our analysis of current and future memory technologies can be used in cookbook fashion to design lookup engines for IPv4 unicast or IPv6 unicast at rates up to OC-192. Coupled with a hardware reference design we present a novel memory management scheme that can be tied with tree bitmap incremental updates to yield a deterministic time, and low memory overhead approach.

We note that our algorithm differs from the Lulea algorithm (which is the only existing algorithm to encode prefix tables as compactly as we do) in several key ways. First, we use a different encoding scheme that relies on two bit maps per node. Second, we use only one memory reference per trie node as opposed to two per trie node in Lulea. Third, we have guaranteed fast update times; in the Lulea scheme, a single update can cause almost the entire table to be rewritten. Fourthly, unlike Lulea, our core algorithm can be tuned to leverage off the structure of modern memories.

The outline of the rest of the paper is as follows. In Section 2, we describe memory models; the memory model is perhaps a research contribution in its own right, and may be useful to other designers. In Section 3, we briefly review previous work, especially Lulea tries [3] and expanded tries [20]. In Section 4, we present the core lookup scheme called Tree Bitmap and contrast it carefully with Lulea and expanded tries. In Section 5, we describe optimizations to the core algorithm that reduce the access size required for a given stride size, and provide deterministic size requirements. In Section 6, we describe reference software implementations. In Section 7, we describe a reference hardware implementation. Sections 6 and 7 are based on the memory model in Section 2 and the core ideas and optimizations of Sections 4 and 5. We conclude in Section 8.

2 Models

2.1 Memory Models

We will consider a series of choices for implementation ranging from pure software to pure hard-

ware. We describe memory models for such choices.

2.1.1 Software

We will consider software platforms using modern processors such as the Pentium and the Alpha. These CPUs execute simple instructions very fast (few clock cycles) but take much longer to make a random access to main memory. The only exception is if the data is in either the Primary (L1) or Secondary Cache (L2) which allow access times of a few clock cycles. The distinction arises because main memory uses DRAM (60 nsec per random access), while cache memory is expensive but fast SRAM (10-20 nsec). When a READ is done to memory of a single word, an entire cache line is fetched into the cache. This is important because the remaining words in the cache line can be accessed cheaply for the price of a single memory READ.

The filling of cache lines exploits a simple fact about Dynamic RAMS that is crucial to an understanding of modern memory systems. While random access to an arbitrary word (say 32 bits) W in a DRAM may take a long time (say 60 nsec), once W is accessed, access to locations that are contiguous to W (burst mode) can take place at much faster speeds comparable to SRAM speeds. This is because DRAMs are often internally structured as a two dimensional memory consisting of rows (often called pages) of several kilobits: to access a word W the entire page containing W must first be retrieved.

2.1.2 On-Chip ASIC Memory

Recall that an ASIC is a special IC for (say) IP lookups. Since much of the memory latency is caused by going off-chip to a separate memory chip and because chip densities have been going up, it makes sense to put as much memory on-chip as possible. Artisian Components [1]advertises memory generators for several ASIC foundries, and can therefore provide a rough picture of the current state of on-chip SRAM technology. At.25 micron they say that a 64kbit block of memory can run at 200 Mhz worst case for large configurations. Devoting half of the die of a large.25 micron die should in our estimation yield at least 4 Mbits of SRAM. On-chip DRAM is becoming a standard option from several ASIC foundries. But it is unclear at this time what will be the real limitations of embedded DRAM, so we will not consider it any more in this paper.

2.1.3 Discrete Memory Technologies

Besides the use of standard SRAM and DRAM, the major off-chip RAM technologies today and in the future are PC-100 SDRAM [7], DDR-SDRAM[6], Direct Rambus[5], and Synchronous SRAM[16]. Efficient use of most of these technologies are based on the idea of memory interleaving to hide memory latency. Consider for example Figure 1 which shows an IP Lookup ASIC that interfaces to a PC-100 SDRAM. The figure shows that the SDRAM internally has two DRAM banks, and the interface between the ASIC and the SDRAM is 80 pins (6 bits for Control, 12 for Address, and 64 bits for data). The basic picture for other technologies is similar except that we could use more than two banks (e.g., 16 banks in RAMBUS).

Notice also that in the figure we show that the IP lookup ASIC has a FIFO buffer of destination addresses. This is so that we can allow some limited pipelining of destination address lookups to fully utilize the memory bandwidth. For example, assume that lookups require 4 accesses to a data structure, starting with a root node that points to a second level node, to a third level node, to a fourth level node. If we have two banks of memories, we can place the first two levels of data structure in the first memory bank and the third and fourth in the second memory bank. If we can simultaneously work on the lookup for two IP addresses (say D1 followed by D2), while we are looking up the third and fourth level for D1 in Bank 2, we can lookup the first and second levels for D2 in Bank 1. If it takes 40 nsec per memory access, it still takes 4 * 40 = 160 nsec to lookup D1. However, we have a higher throughput of two IP lookups every 160 nsec.

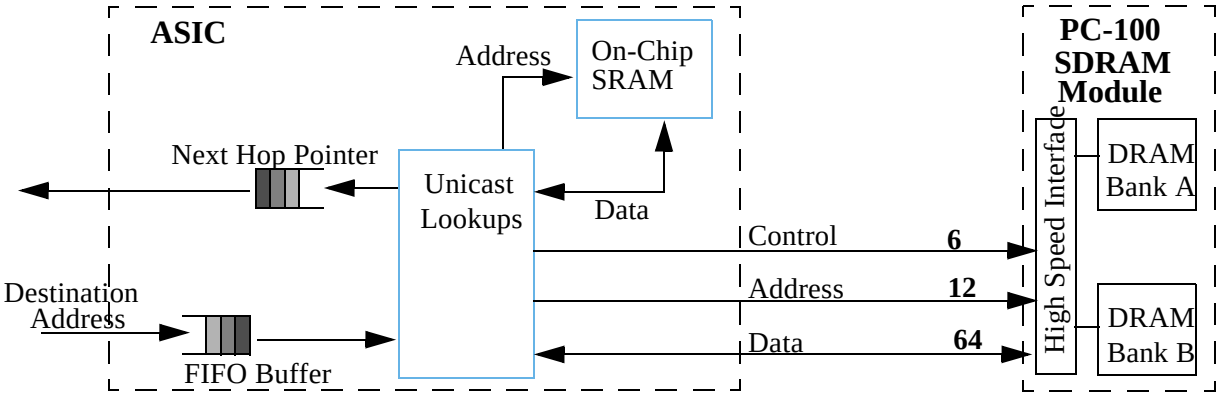

Figure 1: Block Diagram of Lookup Reference Design

Pipelining may appear complex but it is not because the control of the multiple memory banks is done across the ASIC interface pins (as opposed to dealing with the memory through an intermediate memory controller). The user of the lookup algorithm (e.g., the forwarding code) also has a simple interface: an IP lookup answer can be read in a fixed amount of time (160 nsec in the above example) after the request is made. The results (e.g., next hop info) are posted to a FIFO buffer of next hops (see Figure) which are read by the forwarding engine at the appropriate time.

For an algorithm designer, we wish to abstract the messy details of these various technologies into a clean model that can quickly allow us to choose between algorithms. The most important information required is as follows. First, if we wish to have further interleaving for more performance we need more memory interfaces and so more interface pins. Because interface pins for an ASIC are a precious resource, it is important to quantify the number of memory interfaces (or pins) required for a required number of random memory accesses. Other important design criteria are the relative costs of memory per bit, the amount of memory segmentation, and the optimal burst size for a given bus width and random access rate that fully utilizes the memory.

Table 1 shows a table comparing the more quantifiable differences between the discrete memory technology choices. To understand a sample row of this Table, let us go back to the earlier Figure 1 for SDRAMs and use it to obtain the numbers in the first row of the Table. We see from the figure that the ASIC has an 80 pin interface, which is what we put in the first column. The data rate per pin is given by the manufacturer to be 100 Mbps (10 nsec per bit). We see from the figure that the logical number of banks is two. Finally, the manufacturer specs can be used to tell us that a random access to 8 bytes (notice that the data interface has 64 data pins = 8 bytes) takes 40 nsec. Thus we can compute that the number of random memory accesses in 160 nsec is exactly 4 (160/40). Notice that the first five columns have been filled in essentially from manufacturer specs. One cost measure of these random accesses is given in the 6th column by showing the ratio of ASIC pins to random memory accesses. The final column shows the optimal burst sizes when accessing the memory at the highest random rate possible. For the first row, this arises because we can transfer 64-bits of data in 10 ns (data rate is 100 Mbps). Since a random access takes 40 ns (4 accesses in 160 ns), we need to have a burst of four 64-bit words (32-bytes total) to avoid idling memory

Memory Technology with Data path Width	ASIC pins	Data Rate (Mbps)	Logical # of Banks	# of Random Memory Accesses every 160 ns	ASIC Pins/ Random Memory Access	Block Size (bytes)
PC-100 SDRAM (64-bit)	80	100	2	4	20	32
DDR-SDRAM (64-bit)	80	200	4	4	20	64

Table 1: Memory Technology For Hardware IP Lookups

Memory Technology with Data path Width	ASIC pins	Data Rate (Mbps)	Logical # of Banks	# of Random Memory Accesses every 160 ns	ASIC Pins/ Random Memory Access	Block Size (bytes)
Direct Rambus(16-bit)	76	800	8[a]	16	4.75	16
Synchronous SRAM(32-bit)	52	100	1	16	3.25	4

Table 1: Memory Technology For Hardware IP Lookups

a. Direct Rambus actually has 16 banks, but due to the fixed burst size of 8 data ticks, we only need to logically segment the Rambus memory into 8 banks

Why should an IP lookup ASIC designer care about this table? First, the designer can use this table to choose between technologies based on dollar cost and pins/random accesses. Having chosen a technology, the designer can then use the table to determine the optimal burst size. The data structure can then be designed to have every access to the data structure access the optimal burst size. Using bigger burst sizes will be suboptimal (would require more banks or more accesses) and not utilizing the full burst size would use the memory system inefficiently. For example, we observe from the table that the optimal block size for SDRAM based memories (e.g., 32 bytes for PC-100 SDRAM) is much larger than for SRAM based technologies (4 for SRAM). For example, this indicates that we could use larger stride trie algorithms (see later) for DRAM based memories.

2.2 Wire Speed Forwarding

Wire speed IP forwarding has become an industry buzzword that needs to be carefully interpreted as different interpretations differ by as much as a factor of two. The desire for wire speed performance arises because studies show that 50% of all traffic is TCP/IP ack packets [11]. If a router is unable to process minimum length packets, then when a burst of packets arrive, they must be queued for processing at the forwarding engine. Since no information is known about the packets as they await processing, all packets must be treated as equal and best effort and FIFO queueing must be used. This could result in high priority traffic being lost or delayed by a burst of low priority traffic.

Wire speed performance is also desirable since it simplifies system design. To make this precise, we have to examine popular high capacity data link level technologies. For backbone links, the worst case seems to arise with TCP acks running over IP. The link layer technology can be either Gigabit Ethernet[8], IP over PPP over SONET[2], Packet over SONET [13] or IP over ATM over SONET. Simple calculations show that for OC-48 links the worst case forwarding rate requires a lookup every 160 nsec (roughly 6 million packets per second) and climbs up to a forwarding rate of 24 million packets per second for OC-192.

3 Related Work

With new CAM technologies, CAMs in the future may be a fine solution for enterprise and access routers that have less than 32000 prefixes[17]. Thus algorithmic solutions like ours can support much larger prefixes for backbone routers; the flip side is that they need to exhibit some of the determinism in update and lookup times that CAMs provide.

The multi-column lookup scheme [12] uses a B-tree like approach which has reasonably fast lookup times and reasonable storage but poor update times. The binary search on hash tables approach [22] scales very well to IPv6 128 bit addresses but has poor update times and possible non-determinism because of hashed lookups. The Stanford scheme uses a multibit trie with an initial stride of 24 bits, and then two levels of 8 bit tries. Once again updates can be slow in the worst case, despite a set of clever optimizations.

Both the LC-Trie[18] and Expanded Trie schemes [20] allow multibit tries that can have variable strides. In other words, the pointer to a trie code can encode the number of bits sampled at the trie node. The

Expanded trie [20] allows memory to be traded off for lookup speed but requires a fairly high amount of memory (factor of two worse than those of our scheme and the Lulea scheme) despite the use of dynamic programming optimizations. The optimizations can also slow down worst case update times; without considering the time for the optimization program to run, the update times are around 5 msec. The LC-trie fundamentally appears to have slow update times though none are reported.

The Lulea scheme is closest in spirit to ours in that both use multibit tries and bit maps to compress wasted storage in trie nodes. There are fundamental differences, however, which allow our scheme more tunability and faster update times. To show these differences in detail we spend the rest of the section describing three detailed examples of the unibit trie algorithm, the expanded trie algorithm, and the Lulea scheme on a sample prefix database. Since we will use the same sample database to describe our scheme, this should help make the differences clearer.

3.1 Unibit Tries

For updates the one bit at a time trie (unibit trie) is an excellent data structure. The unibit trie for the sample database is shown in Figure 2. With the use of a technique to compress out all one way branches, both the update times and storage needs can be excellent. Its only disadvantage is its speed because it needs 32 random *reads* in the worst case. The resulting structure is somewhat different from Patricia tries which uses a skip count compression method that requires backtracking.

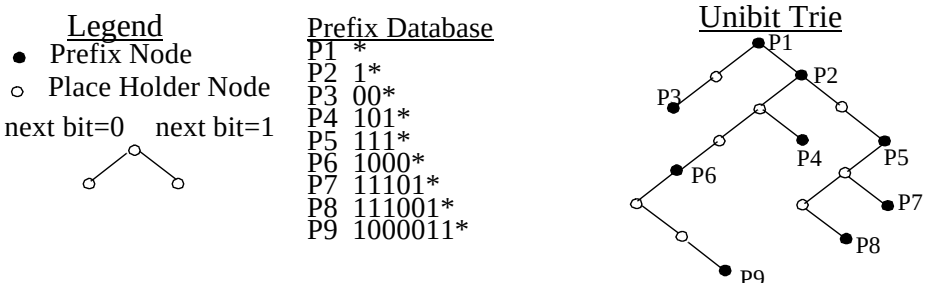

Figure 2: Sample Database with Unibit Trie Representation

3.2 Expanded Tries

In Expanded tries [20], the main idea is to use tries that go several bits at a time (for speed) rather than just 1 bit at a time as in the unibit trie. Suppose we want to do the database in Figure 2 three bits at a time. We will call the number of bits we traverse at each level of the search the “stride length”.

A problem arises with prefixes like P2 = 1* that do fit evenly in multiples of 3 bits. The main trick is to expand a prefix like 1* into all possible 3 bit extensions (100, 101, 110, and 111) and represent P1 by four prefixes. However, since there are already prefixes like P4 = 101 and P5 = 111, clearly the expansions of P1 should get lower preference than P4 and P5 because P4 and P5 are longer length matches. Thus the main idea is to expand each prefix of length that is does fit into a stride length into a number of prefixes that fit into a stride length. Expansion prefixes that collide with an existing longer length prefix are discarded.

Part A of Figure 3 shows the expanded trie for the same database as Figure 2 but using expanded tries with a fixed stride length of 3 (i.e., each trie node uses 3 bits). Notice that each trie node element is a record has two entries: one for the next hop of a prefix and one for a pointer. (Instead of showing the next hops, we have labelled the next hop field with the actual prefix value.) We need to have both pointers and next hop information is for example in entry 100 in the root node. This contains a prefix (P1 = 100) and must also contain a pointer to the trie node containing P6 and pointing to P9.

Notice also the down side of expansion. Every entry in each trie node contains information. For

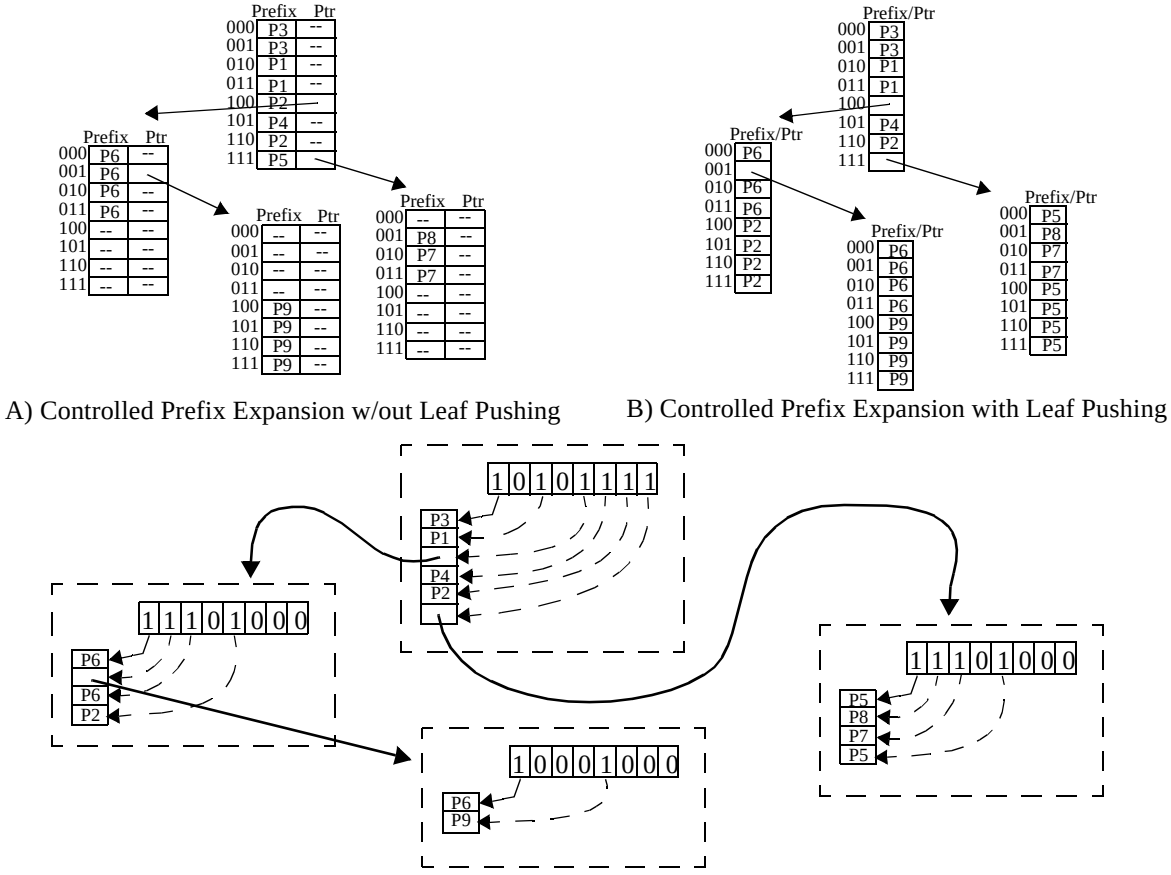

Figure 3: Related Work

example, the root trie node has its first two elements filled with expansions of P3 and the next two with expansions of P1. In general, using larger strides requires the use of larger trie nodes with more wasted space. Reference [20] does show how to use variable stride tries (where the strides are picked to reduce memory) while keeping good search times. However, the best case storage requirements for the Mae East database (after the most sophisticated optimizations) are around 500 Kbytes.

3.3 Lulea

The Lulea scheme [3] does much better in terms of worst case storage, using only 200 Kbytes of memory to store the current Mae East database. One way to describe the scheme is to first subject the expanded trie scheme to the following optimization that we call "leaf pushing". The idea behind leaf pushing is to cut in half the memory requirements of expanded tries by making each trie node entry contain *either* a pointer *or* next hop information but not both. Thus we have to deal with entries like 100 in the root node of Part A of Figure 3 that have both a pointer and a next hop. The trick is to push down the next hop information to the leaves of the trie. Since the leaves do not have a pointer, we only have next hop information at leaves [20]. This is shown in Part B of Figure 3. Notice that the prefix P2 associated with the 100 root entry in Part A has been pushed down to several leaves in the node containing P6.

The Lulea scheme starts with a conceptual leaf pushed expanded trie and (in essence) replaces all consecutive elements in a trie node that have the same value with a single value. This can greatly reduce the number of elements in a trie node. To allow trie indexing to take place even with the compressed nodes, a

bit map with 0's corresponding to the removed positions is associated with each compressed node.

For example consider the root node in Figure 3. After leaf pushing the root node has the sequence of values P3, P3, P1, P1, ptr1, P4, P2, ptr2 (where ptr1 is a pointer to the trie node containing P6 and ptr2 is a pointer to the trie node containing P7). After we replace each string of consecutive values by the first value in the sequence we get P3, -, P1, -, ptr1, P4, P2, ptr2. Notice we have removed 2 redundant values. We can now get rid of the original trie node and replace it with a bit map indicating the removed positions (10101111) and a compressed list (P3, P1, ptr1, P4,P2, ptr2). The result of doing this for all three trie nodes is shown in Part C of the figure.

The search of a trie node now consists of using a number bits specified by the stride (e.g., 3 in this case) to index into each trie node starting with the root, and continuing until a null pointer is encountered. On a failure at a leaf, we need to compute the next hop associated with that leaf. For example, suppose we have the data structure shown in Part C and have a search for an address that starts with 111111. We use the first three bits (111) to index into the root node bit map. Since this is the sixth bit set (we need to count the bits set before a given bit), we index into the sixth element of the compressed node which is a pointer to the right most trie node. Here we use the next 3 bits (also 111) to index into the eight bit. Since this bit is a 0, we know we have terminated the search but we must still retrieve the best matching prefix. This is done by counting the number of bits set before the eighth position (4 bits) and then indexing into the 4th element of the compressed trie node which gives the next hop associated with P5.

The Lulea scheme is used in [3] for a trie search that uses strides of 16,8, and 8. Without compression, the initial 16 bit array would require 64K entries of at least 16 bits each, and the remaining strides would require the use of large 256 element trie nodes. With compression and a few additional optimizations, the database fits in the extremely small size of 200 Kbytes.

Fundamentally, the implicit use of leaf pushing (which ensures that only leaves contain next hop information) makes insertion inherently slow in the Lulea scheme. Consider a prefix P0 added to a root node entry that points to a sub-trie containing thirty thousand leaf nodes. The next hop information associated with P0 has to be pushed to thousands of leaf nodes. In general, adding a single prefix can cause almost the entire structure to be modified making bounded speed updates hard to realize.

If we abandon leaf pushing and start with the expanded trie of Part A of Figure 3, when we wish to compress a trie node we immediately realize that we have two types of entries, next hops corresponding to prefixes stored within the node, and pointers to sub-tries. Intuitively, one might expect to need to use TWO bit maps, one for internal entries and one for external entries. This is indeed one of the ideas behind the tree bit map scheme we now describe.

4 Tree Bitmap Algorithm

In this section, we describe the core idea behind our new algorithm that we call Tree Bitmap. Tree Bitmap is a multibit trie algorithm that allows fast searches (one memory reference per trie node unlike 3 memory reference per trie node in Lulea) and allows much faster update times than existing schemes (update times are bounded by the size of a trie node; since we only use trie nodes of stride of no more than 8, this requires only around 256 + C operations in the worst case where C is small). Its also has memory storage requirements comparable (and sometimes slightly better) when compared to the Lulea scheme.

The Tree Bitmap design and analysis is based on the following observations:

* A multibit node (representing multiple levels of unibit nodes) has two functions: to point at children multibit nodes, and to produce the next hop pointer for searches in which the longest matching prefix exists within the multibit node. It is important to keep these purposes distinct from each other.

* With burst based memory technologies, the size of a given random memory access can very large

(e.g, 32 bytes for SDRAM, see Section 2.2). This is because while the random access rate for core DRAMs have improved very slowly, high speed synchronous interfaces have evolved to make the most of each random access. (See Section 2.2). Thus the trie node stride sizes can be determined based on the optimal memory burst sizes.

* Hardware can processes complex bitmap representations of up to 256 bits in a single cycle. Additionally, the mismatch between processor speeds and memory access speeds have become so high that even software can do extensive processing on bitmaps in the time required for a memory reference.

* To keep update times bounded it is best not to use large trie nodes (e.g., 16 bit trie nodes used in Lulea). Instead, we use smaller trie nodes (at most 8 bits). Any small speed loss due to the smaller strides used is offset by the reduced memory access time per node (1 memory access per trie node versus three in Lulea).

* To insure that a single node is always retrieved a single page access, nodes should always be power of 2 in size and properly aligned (8 byte nodes on 8-byte boundaries etc.) on page boundaries corresponding to the underlying memory technology.

Based on the these observations, the Tree Bitmap algorithm is based on four key ideas.

The first idea in the Tree Bitmap algorithm is that all child nodes of a given trie node are stored contiguously. This allows us to use just one pointer for all children (the pointer points to the start of the child node block) because each child node can be calculated as an offset from the single pointer. This can reduce the number of required pointers by a factor of two compared with standard multibit tries. More importantly it cuts down the size of trie nodes, as we see below. The only disadvantage is that the memory allocator must, of course, now deal with larger and variable sized allocation chunks. Using this idea, the same 3-bit stride trie of Figure 3 is redrawn as Figure 4.

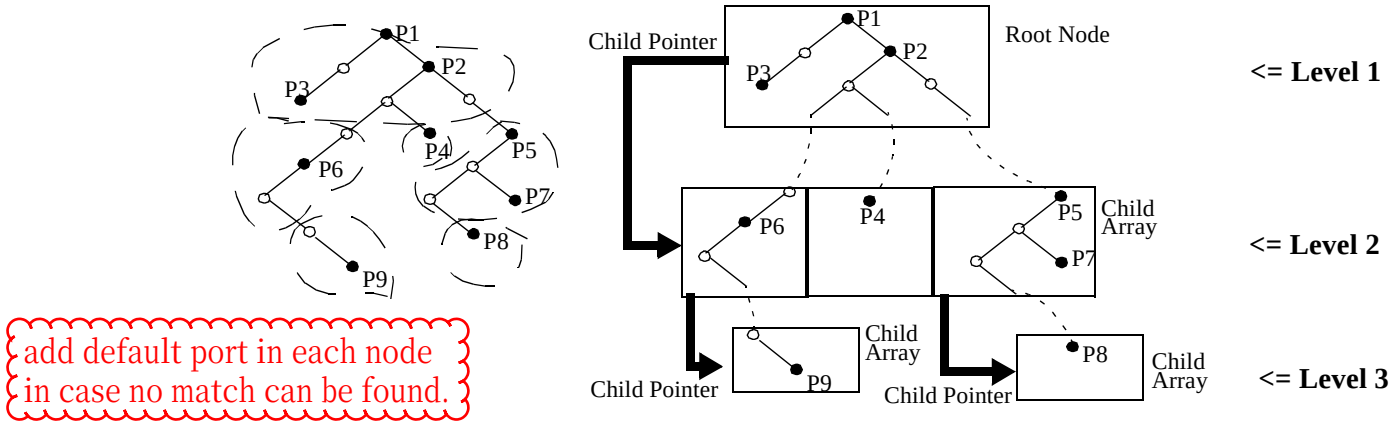

Figure 4: Sample Database with Tree Bitmap

The second idea is that there are two bit maps per trie node, one for all the internally stored prefixes and one for the external pointers. See Figure 5 for an example of the internal and external bit maps for the root node. The use of two bit maps allows us to avoid leaf pushing. The internal bit map is very different from the Lulea encoding, and has a 1 bit set for every prefixes stored within this node. Thus for an r bit trie node, there are $2^{\{r-1\}}$ possible prefixes of lengths $< r$ and thus we use a 2^r-1 bit map.

For the root trie node of Figure 4, we see that we have three internally stored prefixes: P1 = *, P2 = 1*, and P3 = 00 *. Suppose our internal bit map has one bit for prefixes of length 0, two following bits for prefixes of length 1, 4 following bits for prefixes of length 2 etc. Then for 3 bits the root internal bit map becomes 1011000. The first 1 corresponds to P1, the second to P2, the third to P3. This is shown in Figure

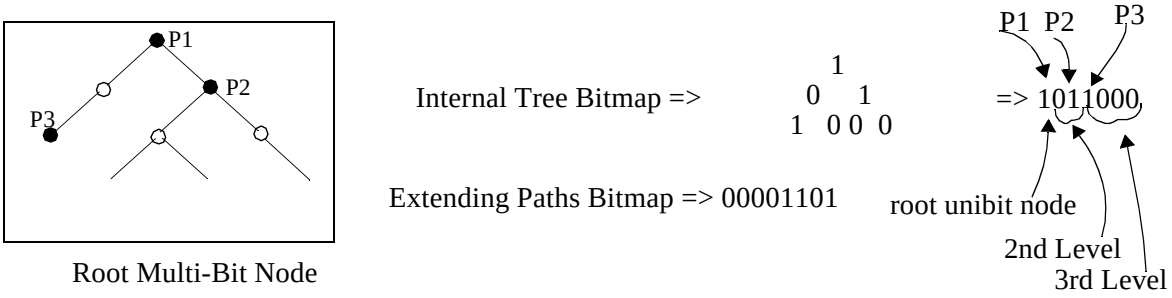

Figure 5: Multibit Node Compression with Tree Bitmap

5.

The external bit map contains a bit for all possible 2^r child pointers. Thus in Figure 4, we have 8 possible leaves of the 3 bit subtrie. Only the fifth, sixth, and eight leaves have pointers to children. Thus the extending paths (or external) bit map shown in Figure 5 is 00011001 As in the case of Lulea, we need to handle the case where the original pointer position contains a pointer and a stored prefix (e.g., location 111 which corresponds to P5 and also needs a pointer to prefixes like P7 and P8. The trick we use is to push all length 3 prefixes to be stored along with the zero length prefixes in the next node down. For example, in Figure 4, we push P5 to be in the right most trie node in Level 2. In the case of P4 (which was stored in the root subtrie in the expanded trie of Figure 3a) we actually have to create a trie node just to store this single zero length prefix.

The third idea is to keep the trie nodes as small as possible to reduce the required memory access size for a given stride. Thus a a trie node is of fixed size and only contains an external pointer bit map, an internal next hop info bit map, and a single pointer to the block of child nodes. But what about the next hop information associated with any stored prefixes?

The trick is to store the next hops associated with the internal prefixes stored within each trie node in a *separate* array associated with this trie node. For memory allocation purposes result arrays are normally an even multiple of the common node size (e.g. with 16-bit next hop pointers, and 8-byte nodes, one result node is needed for up to 4 next hop pointers, two result nodes are needed for up to 8, etc.) Putting next hop pointers in a separate result array potentially requires two memory accesses per trie node (one for the trie node and one to fetch the result node for stored prefixes). However, we use a simple lazy strategy to not access the result nodes till the search terminates. We then access the result node corresponding to the last trie node encountered in the path that contained a valid prefix. This adds only a single memory reference at the end besides the one memory reference required per trie node.

The search algorithm is now quite simple. We start with the root node and use the first bits of the destination address (corresponding to the stride of the root node, 3 in our example) to index into the external bit map at the root node at say position P. If we get a 1 in this position there is a valid child pointer. We count the number of 1s to the left of this 1 (including this 1) as say I. Since we know the pointer to the start position of the child block (say C) and the size of each trie node (say S), we can easily compute the pointer to the child node as C + (I * S).

Before we move on to the child, we also check the internal bit map to see if there is a stored prefix corresponding to position P. This requires a completely different calculation from the Lulea style bit calculation. To do so, we can imagine that we successively remove bits of P starting from the right and index into the corresponding position of the internal bit map looking for the first 1 encountered. For example, suppose P is 101 and we are using a 3 bit stride at the root node bit map of Figure 5. We first remove the right most bit, which results in the prefix 10*. Since 10* corresponds to the sixth bit position in the internal bit map, we check if there is a 1 in that position (there is not in Figure 5). If not, we need to remove the right most

two bits (resulting in the prefix 1*). Since 1* corresponds to the third position in the internal bit map, we check for a 1 there. In the example of Figure 5, there is a 1 in this position, so our search ends. (If we did not find a 1, however, we simply remove the first 3 bits and search for the entry corresponding to * in the first entry of the internal bit map.).

This search algorithm appears to require a number of iterations proportional to the logarithm of the internal bit map length. However, in hardware for bit maps of up to 512 bits or so, this is just a matter of simple combinational logic (which intuitively does all iterations in parallel and uses a priority encoder to return the longest matching stored prefix). In software this can be implemented using table lookup. (A further trick using a stored bit in the children further avoids doing this processing more than once for software; see software reference implementation.) Thus while this processing appears more complex than the Lulea bit map processing it is actually not an issue in practice.

Once we know we have a matching stored prefix within a trie node, we do not immediately retrieve the corresponding next hop info from the result node associated with the trie node. We only count the number of bits before the prefix position (more combinational logic!) to indicate its position in the result array. Accessing the result array would take an extra memory reference per trie node. Instead, we move to the child node while remembering the stored prefix position and the corresponding parent trie node. The intent is to remember the last trie node T in the search path that contained a stored prefix, and the corresponding prefix position. When the search terminates (because we encounter a trie node with a 0 set in the corresponding position of the external bit map), we have to make one more memory access. We simply access the result array corresponding to T at the position we have already computed to read off the next hop info.

Figure 6 gives pseuducode for full tree bitmap search. It assumes a function treeFunction that can find the position of the longest matching prefix, if any, within a given node by consulting the internal bitmap (see description above). "LongestMatch" keeps track of a pointer to the longest match seen so far. The loop terminates when there is no child pointer (i.e., no bit set in External bit map of a node) upon which we still have to do our lazy access of the result node pointed to by LongestMatch to get the final next hop. We assume that the address being searched is already broken into strides and stride[i] contains the bits corresponding to the i'th stride.

```
1.    node:= root;  (* node is the current trie node being examined; so we start with root as the first trie node *)
1.    i:= 1;   (* i is the index into the stride array; so we start with the first stride *)
1.    do forever
1.      if (treeFunction(node.internalBitmap,stride[i]) is not equal to null) then
1.                  (* there is a longest matching prefix, update pointer *)
1.        LongestMatch:= node.ResultsPointer + CountOnes(node.internalBitmap,
1.                treeFunction(node.internalBitmap, stride[i]));
1.      if (externalBitmap[stride[i]] = 0) then    (* no extending path through this trie node for this search *)
1.          NextHop:= Result[LongestMatch]; (* lazy access of  longest match pointer to get next hop pointer *)
1.          break; (* terminate search)
1.       else (* there is an extending path, move to child node *)
1.          node:= node.childPointer + CountOnes(node.externalBitmap, stride[i]);
1.          i=i+1; (* move on to next stride *)
1.    end do;
```

Figure 6: Tree Bitmap Search Algorithm for Destination Address whose bits are in an array called stride

So far we have implicitly assumed that processing a trie node takes one memory access. This can be done if the size of a trie node corresponds to the memory burst size (in hardware) or the cache line size (in software). That is why we have tried to reduce the trie node sizes as much as possible in order to limit the number of bits accessed for a given stride length. In what follows, we describe some optimizations that reduce the trie node size even further.

5 Optimizations

For a stride of 8, the data structure for the core algorithm would require 255 bits for the Internal Bitmap, 256 bits for the External Bitmap, 20 bits for a pointer to children, 20 bits for a result pointer which is 551. Thus the next larger power of 2 node size is 1024 bits or 128-bytes. From our table in Section 2.1. we can see that for some technologies this would require several memory interfaces and so drive up the cost and pin count. Thus we seek optimizations that can reduce the access size.

5.1 Initial Array Optimization segmentation table

Almost every IP lookup algorithm can be speeded up by using an initial array (e.g., [3],[14],[22]). Array sizes of 13 bits or higher could, however, have poor update times. A example of initial array usage would be an implementation that uses a stride of 4, and an initial array of 8-bits. Then the first 8-bits of the destination IP address would be used to index into a 256 entry array. Each entry is a dedicated node possibly 8-bytes in size which results in a dedicated initial array of 2k bytes. This is a reasonable price in bytes to pay for the savings in memory accesses. In a hardware implementation, this initial array can be placed in on chip memory.

5.2 End Node Optimization

We have already seen an irritating feature of the basic algorithm in Figure 4. Prefixes like P4 will require a separate trie node to be created (with bitmaps that are almost completely unused). Let us call such nodes as "null nodes". While this cannot be avoided in general, it can be mitigated by picking strides carefully. In a special case, we can also avoid this waste completely. Suppose we have a trie node that only has external pointers that point to "null nodes". Consider Figure 7 as an example that has only one child which is a "null node". In that case, we can simply make a special type of trie node called an endnode (the type of node can be encoded with a single extra bit per node) in which the external bit map is eliminated and substituted with an internal bit map of twice the length. The endnode now has room in its internal bit map to indicate prefixes that were previously stored in "null nodes". The "null nodes" can then be eliminated and we store the prefixes corresponding to the null node in the endnode itself. Thus in Figure 7, P8 is moved up to be stored in the upper trie node which has been modified to be an endnode with a larger bit map.

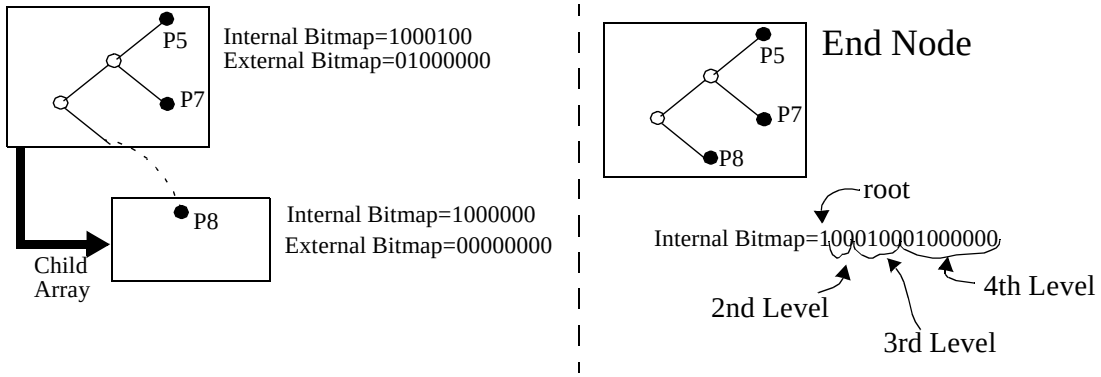

Figure 7: End Node Optimization

5.3 Split Tree Bitmaps

Keeping the stride constant, one method of reducing the size of each random access is to split the internal and external bitmaps. This is done by placing only the external bitmap in each "Trie" node. If there

is no memory segmentation, the children and the internal nodes from the same parent can be placed contiguously in memory. If memory segmentation exists, it is a bad design to have the internal nodes scattered across multiple memory banks. In the case of segmented memory, one option is to have the trie node point at the internal node, and the internal node point at the results array. Or the trie node can have three pointers to the child array, the internal node, and to the results array. Figure 8 shows both options for pointing to the results array and implementing split tree bitmap.

To make this optimization work, each child must have a bit indicating if the parent node contains a prefix that is a longest match so far. If there was a prefix in the path, the lookup engine records the location of the internal node (calculated from the data structure of the last node) as containing the longest matching prefix thus far. Then when the search terminates, we first have to access the corresponding internal node and then the results node corresponding to the internal node. Notice that the core algorithm accesses the next hop information lazily; the split tree algorithm accesses even the internal bit map lazily. What makes this work is that any time a prefix P is stored in a node X, all children of X that match P can store a bit saying that the parent has a stored prefix. The software reference implementation uses this optimization to save internal bit map processing; the hardware implementations use it only to reduce the access width size (because bit map processing is not an issue in hardware).

A nice benefit of split tree bitmaps is that if a node contained only paths and no internal prefixes, a null internal node pointer can be used and no space will be wasted on the internal bitmap. For the simple tree bitmap scheme with an initial stride of 8 and a further stride of 4, 50% of the 2971 trie nodes in the Mae-East database do not have an internally terminating prefixes..

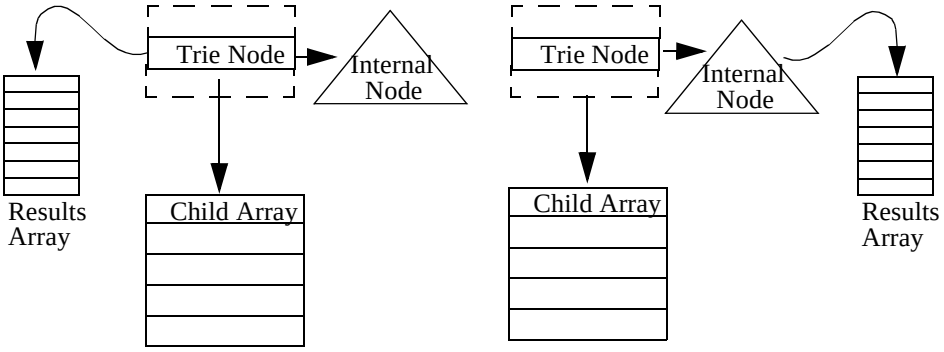

Figure 8: Split Tree Bitmap Optimization

5.4 Segmented Bitmaps

After splitting the internal and external bitmaps into separate nodes, the size of the nodes may still be too large for the optimal burst size (see 2.1). The next step for reducing the node sizes is to segment the multi-bit trie represented by a multi-bit node. The goal behind segmented bitmaps is to maintain the desired stride, keep the storage space per node constant, but reduce the fetch size per node. The simplest case of segmented bitmaps is shown in Figure 9 with a total initial stride of 3. The subtrie corresponding to the trie node is split into its 2 child subtries, and the initial root node duplicated in both child subtries. Notice that the bit maps for each child subtrie is half the length (with one more bit for the duplicated root). Each child subtrie is also given a separate child pointer as well as a bit map and is stored. as a separate "segment". Thus each trie node contains two contiguously stored segments.

Because each segment of a trie node has its pointers, the children and result pointers of other segmented nodes are independent. While the segments are stored contiguously, we can use the high order bits of the bits that would normally have been used to access the trie node to access only the required segment. Thus we need to roughly access only half the bit map size. For example, using 8 bit strides, this could

reduce the bit map accessed from 256 bits to 128 bits.

When this simple approach is extended to multiple segmentations, the complexity introduced by the segmented bitmap optimization is that if k is the initial stride, and 2^j is the number of final segmented nodes, the internal prefixes for the top k-j rows of prefixes are shared across multiple segmented nodes. The simplest answer is to simply do controlled prefix expansion and push down to each segmented node the longest matching result from the top k-j rows.

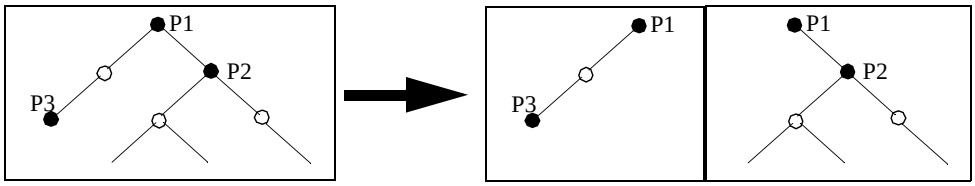

Figure 9: Segmented Tree Bitmap

5.5 CAM Nodes

Empirical results show that a significant number of multi-bit nodes have only a few internal prefixes. In these cases the space normally occupied by internal bitmaps and a pointer to a results arrays can be replaced by simple CAM type entries that have match bits, match length, and next hop pointers. The gain is that the next hop pointers are in the CAM nodes and not in a separate results array taking up space. For end nodes and internal nodes, even single entry CAM nodes was found to typically result in over half of the next hop pointers moving from results arrays, to inside CAM end nodes. There are quickly diminishing returns however.

6 Software Reference Design

Our software reference implementation of the TreeBitmap algorithm, which we present in this section, outperforms other state-of-the-art IP address lookup algorithms in some dimension of interest: these include lookup speed, or insertion/deletion speed, and total size. At the same time the implementation of the algorithm is very simple.

For software, we chose a stride of 4; by keeping this small, we are able to fit the entire node into 8 bytes, which is the size of an internal cache line on many processors. Also, with a small stride, the bitmaps are small: the internal tree bitmap is 15 bits, and the external paths bitmap is 16 bits. With such small bitmaps, we can use a byte lookup table to efficiently compute the number of set bits in a bitmap. We also eliminate the pointer to a results array by allocating children nodes as well as the results in a contiguous buffer, so that only a single pointer can be used to access both arrays with children to the right and results to the left, Thus either array can be accessed directly, without having to know the number of entries in the other array. We also add a "parent_has_match" bit to each node. This bit allows us to avoid searching for a matching prefix in all but two of the internal bitmaps of encountered node. During trie traversal we use this bit to determine the last node that had such a valid prefix and lazily search the prefix bit map of that node after the search terminates.

Our reference implementation uses an initial 8K entry lookup table and a 4 bit stride to create a 13,4,4,4,4,3 trie. Since there is a large concentration of prefixes at lengths 16 and 24, we ensure that for the vast majority of prefixes the end bits line up at the bottom of a trie node, with less wasted space. This explains the 13,4,4,... ordering: 13+4 is 17, which means the 16 bit prefixes will line up on the lowest level of the tree_bitmap in the nodes retrieved from the initial table lookup. Similarly, 13+4+4+4 is 25, so once

again the pole at 24 will line up with the lowest level of the tree_bitmap in the third level nodes of the trie.

6.1 Software Performance Results

We ran several experiments using our reference implementation on two different platforms: a 296MHz SPARC Ultra-II with 2 MB of cache and a 300MHz Pentium-II PC with 512 KB of L2 cache. All our experiments used several publicly available prefix databases from public NAPs. The results are summarized in Table 2 .

The first two columns of the table contain information about the prefix database that was used and the number of prefixes in the database. The remaining columns in the table are measurements of the average time for lookups and for insertion of prefixes in the forwarding database for the two experimental platforms. Three sets of lookup speeds are reported. In the fifth column, we ran a sequence of randomly generated addresses through the database. Since a considerable number of the randomly generated addresses do not match any address prefix this results in very high (and unrealistic) lookup rates in excess of 10 million pps. We have presented this set of results here because other researchers have used this as a metric of lookup performance [12].

The sixth column presents results from running a real packet trace through the database. These results represent what we would actually observe in practice, and include effects of improved cache performance resulting from the locality of the traces; the corresponding packets- per-second rate ranges from 5.2 to 8.7 million on the UltraSPARC, and from 4 to 6.5 million on the Pen- tium-II. Finally, for the last lookup speed (seventh) column, we chose prefixes at random from the prefix database itself, and fed these prefixes (with trailing 0s) into the database as addresses that need to be looked up. Since the prefixes were chosen from the database, every address fed into the database would match some prefix, and since they were selected at random, cache and locality effects are suppressed. This explains why the numbers in this column are larger than those in the packet trace column. These results, which range from 2.8 to 3.8 million pps, represent what we would expect to get in a worst-case scenario where packets exhibit no locality in their destination address field. Finally, the last column of the table lists the average insertion times for a prefix, computed as the total time to build the database divided by the number of prefixes.

Prefix Database Name	No. of prefixes in database	Total Allocated Memory	Machine Type	Avg. Lookup Time (random)	Avg. Lookup Time (pkt. trace)	Avg. Lookup Time (Database)	Avg. Insert Time
MAE East	40,902	312 KB	Ultra-II	87 ns	187 ns	280 ns	2.7 µs
			Pentium	95 ns	249 ns	348 ns	3.5 µs
MAE-Weat	19,229	176 KB	Ultra-II	82 ns	182 ns	271 ns	2.7 µs
			Pentium	84 ns	240 ns	325 ns	3.5 µs

Table 2: Results from Software Reference Implementation

7 Hardware Reference Design

In this section we investigate the design of a hardware lookup engine based on the Tree Bitmap algorithm. The goal here is not to present comprehensive design details, but to discuss an example design and illustrate the interesting features that the tree bitmap algorithm provides to a hardware implementation. The issues discussed include the base pipelined design, hardware based memory management, incremental updates, and finally an analysis of the design. An area not explored here due to space limitations is the ability to use tree bitmap, coupled with the presented memory management scheme and path compression to give impressive worst case storage bounds.

7.1 Design Overview

The reference design presented in this chapter is a single chip implementation of an IP Lookup engine using embedded SRAM. The first target of this design is deterministic lookup rates supporting worst cast TCP/IP over OC-192c link rates (25 million lookups per second). Another target is high speed and deterministic update rates transparent to the lookup processes and easily handled in hardware. A final goal is an memory management integrated with updates (so easily handled in hardware) and able to provide high memory utilization.

Below in Figure 10 is a block diagram of the IP Lookup Engine. The left side of the engine has the lookup interface with a 32-bit search address, and a return next hop pointer. The lookup engine is pipelined so there will be multiple addresses evaluated simultaneously. On the right side of the lookup engine is an update interface. Updates take the form: type {insert, change, delete}, prefix, prefix length, and for insertions or table changes a Next Hop Pointer. When an update is completed the Lookup Engine returns an acknowledgment signal to conclude the update.

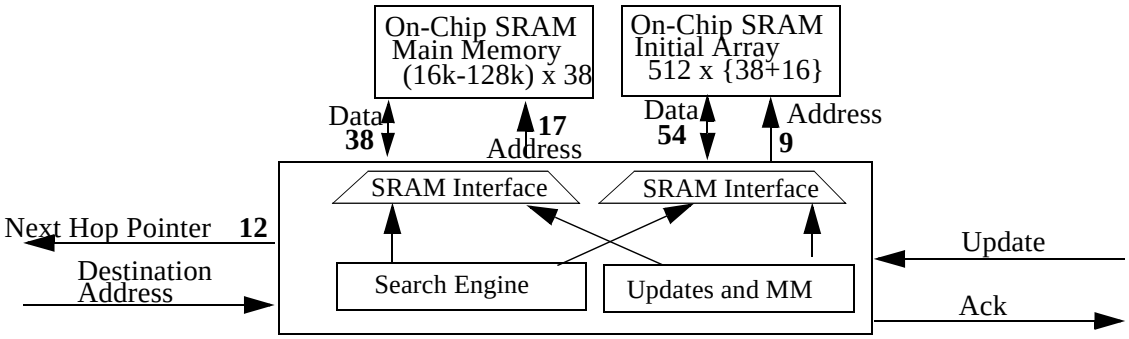

Figure 10: Block Diagram of IP Lookup Engine Core

Looking inside the block diagram of Figure 10, there are two SRAM Interface blocks. The right SRAM Interface block connects to an SRAM containing the initial array. The initial array optimization (Section 5.1) trades off a permanently allocated memory for a reduction in the number of memory accesses in the main memory. For the reference design the initial array SRAM is 512 entries and 54 bits wide (this contains a trie node and a 12-bit next hop pointer). The left SRAM Interface block connects to the main memory of the lookup engine. The reference design has a 38 bit wide interface to the main memory and supports implementations of up to 128k memory entries in depth. It is important to note that all addressing in the main memory is done relative to the node size of 38 bits.

The design parameters based on the terminology of previous chapters are: 9 bit initial stride (So first 9 bits of search address used to index into initial array), 4 bit regular stride, Split Tree Bitmap Optimization, End nodes (Note the lowest level of the trie must use end nodes and will encompass the last 3 search address bits), and CAM optimization for end nodes and internal nodes.

Each node first has a type (trie, end, or cam), and for the basic trie node there is a extending bitmap and a child address pointer. Additionally there are a couple extra bits for various optimizations like

skip_length for path compression and parent_has_match to flag a match in the parent.

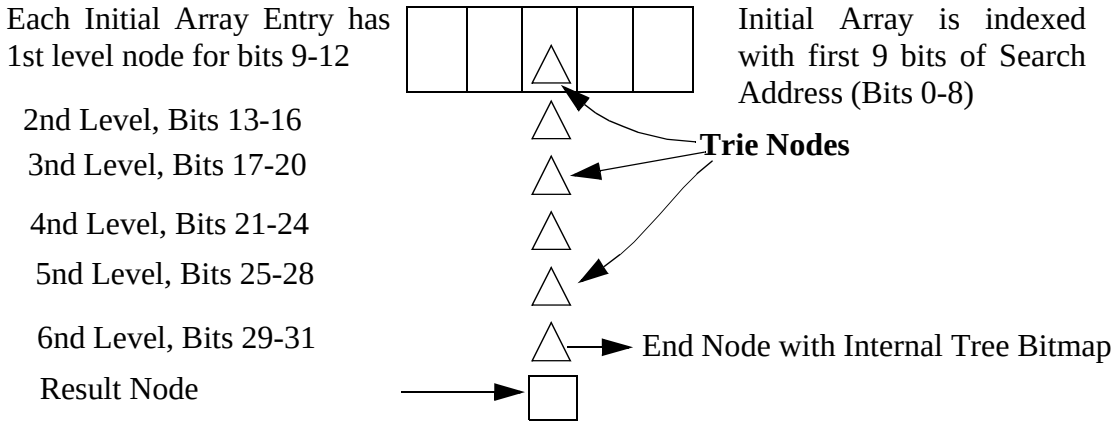

Figure 11: Node Sequence For a full 32 bit search

Figure 11 illustrates an example longest node access pattern for the reference design. In this figure, the first 9 bits of the search address (bits 0 through 8) are used to index into the initial array memory. At the indexed location is stored a node (in this illustration it is a trie node) which represents bits 9-12 of the search address. After the first trie node is fetched from the initial array, 4 trie nodes and then an end node are fetched from the main memory. The end node points at a result node which contains the next hop pointer for this example search.

In the worst case there are 7 memory accesses per lookup slot for search accesses. Devoting 1 out of every 8 memory accesses to control operations (updates and memory management), there are 8 memory accesses required per lookup. At 200 Mhz, with full pipelining the lookup rate will be deterministically 25 million per second.

7.2 Memory Management

A very important part of an IP lookup engine design is the handling of memory management. A complex memory management algorithm requires processor management of updates which is expensive and hard to maintain. Poor memory utilization due to fragmentation can negate any gains made by optimizations of the lookup algorithm itself for space savings. Memory management is therefore a very important part of a lookup engine and requires careful consideration.

For variable length allocation blocks (which most compression trie schemes require) the memory management problem is more difficult than for fixed sized allocation blocks. The simplest method of handling a small fixed number of possible lengths, is to segment memory such that there is a separate memory space for each allocation block size. Within each memory space a simple list based memory management technique is used. The problem with this approach is that it is possible that one memory space will fill while another memory space goes very under-utilized. This means that the critical point at which a filter insertion fails can happen with total memory utilization being very low. A possible way to avoid the under utilization is to employ programmable pointers that divide the different memory spaces. A requirement of this technique is an associated compaction process that keeps nodes for each bin packed to allow pointer movement. With perfect compaction, the result is perfect memory management in that a filter insertion will only fail when memory is 100% utilized.

Compaction without strict rules is difficult to analyze, difficult to implement, and does not provide any guarantees with regards to update rates or memory utilization at the time of route insertion failure. What is needed is compaction operations that are tied to each update such that guarantees can be made. Presented here is a new and novel memory management algorithm that uses programmable pointers, and only does compaction operations in response to an incremental update (*reactive compaction)*. This approach is

most effective with a limited number of allocation blocks which would result from small strides.

For the reference design, there are 17 different possible allocation block sizes. The minimum allocation block is 2 nodes since each allocation block must have an allocation block header. The maximum allocation block size is 18 nodes. This would occur when you have a 16 node child array, an allocation header node, and an internal node of the parent. The memory management algorithm presented here first requires that all blocks of each allocation block size be kept in a separate contiguous array in memory. A begin and end pointer bounds the 2 edges of a given memory space. Memory spaces for different block sizes never inter-mingle. Therefore, memory as a whole will contain 17 memory spaces, normally with free space between the memory spaces (it is possible for memory spaces to abut). To fully utilize memory we need to keep the memory space for each allocation block size tightly compacted with no holes. For the reference design there are 34 pointers total, 17 α pointers representing the 'top' of a memory space and 17 β pointers representing the 'bottom' of a memory space. The first rule for allocating a new free block is that when an allocation block of a given size is needed, check if there is room to extend the size of the memory space for that block size either up or down. After allocating the new node, adjust the appropriate end pointer (either α or β) for the memory space. Second, if the memory space cannot be extended either up or down, a linear search is done in both directions until a gap is found between two memory spaces large enough for the desired allocation size. For whatever distance from the free space to the memory space needing an allocation, the free space will have to be passed by copying the necessary number of allocation block in each memory space from one end to the other. For each memory space the free space must be passed through, this might require copying one or more allocation blocks from one end to the other and adjusting its pointers.

Figure 12 shows an example of a portion of memory space near the top of main memory with memory spaces for allocation block sizes of 2, 3, and 4 nodes.

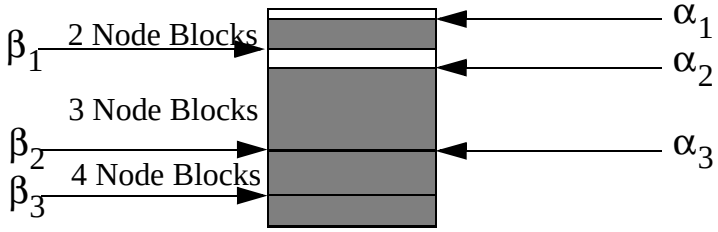

Figure 12: Programmable Pointer Based Memory Management

For the example shown in Figure 12, if an allocation is needed of a block that is 4 nodes in size, it will need the free space between α_2 and β_1 (we will assume the free space is at least 6 nodes in size). To do this allocation, two blocks from the 3 node memory space will be moved from the bottom of the 3 node memory space to the top of that space. Then α_2 and β_2 will be shifted UP 6 memory locations and α_3 will be shifted UP 4 memory locations making room for the newly allocated block. Note that 2 nodes are left between the memory space for 3 and 4 nodes.

For the reference design, the worst case allocation scenario is that the 18 node memory space runs out of adjacent free space, and all the free space remaining is at the other end of memory between the 2 node memory space and the "top" of memory. With the assumption that memory spaces are organized in order {allocation block size of 2 nodes, 3 nodes,..., 18 nodes} then for this worst case scenario 34 nodes of free space must be passed from the far side of the 2 node memory space to the 18 node memory space. The reason that 34 free nodes must be passed from end to end rather then the exact 18 nodes needed is that as the free space propagates through memory, the larger nodes (9-17) must still copy 2 nodes from one end of their memory space to the other to pass at least 18 nodes of free space. A very simple upper bound (overtly conservative but easy to calculate) on the number of memory accesses to allocate a block can be found by

counting the number of memory accesses to move 34 nodes across all of memory, and adding six memory accesses for every block moved to account for the search and modify operation for the parent of each block moved (explained more below). This results in 1852 memory accesses, which means that if dedicating 1 out of 8 memory access to updates, the worst case memory allocation time is 74 microseconds.

One of the problems with compaction mentioned above is that when an allocation block is moved in memory, the parent of the allocation block must have it's pointer modified to reflect the new position. One method for finding a parent is to put with every allocation block the search value that would be used to get to this node. To find the parent of a block, a search is performed until the parent of a node being moved is found and modified. A secondary use of this header block is for the efficient implementation of path compression. There can be an arbitrary skip between parent and child since at the bottom of the search, a complete address for the block containing the final node is checked.

Deallocation of blocks is very simple. For the majority of the cases the deallocated block will be inside the memory space and not on the edge. For this typical case, simply copy one of the blocks at the edge of the memory space into the now vacant block. Then adjust the appropriate end pointer to reflect the compressed memory space. If the deallocated block is on the edge of the memory space then all that needs to be done is pointer manipulation.

7.3 Updates

For all update scenarios, the allocation time will dominate. In the worst case a block of the maximum number of nodes and a block with 2 nodes need to be allocated. In the discussion of memory management we said that 74 microseconds was the worst case allocation time for a maximum size node. It can similarly be shown that 16 microseconds is the maximum for a 2 node allocation block. Given the small number of memory accesses for the update itself, 100 microseconds is a conservative upper bound for a worst case prefix insertion. This means that a deterministic updaterate of 10,000 per second could be supported with this design. Also note that since the updates are incremental, hardware can be designed to handle the updates from a very high level filter specification. This lowers the bandwidth necessary for updates and lends itself to atomic updates from the lookup engine perspective; hardware can play use mechanisms to bypass lookups from fetching data structures that are currently being altered.

7.4 Reference Design Empirical Results

This section explores the empirical results of mapping five databases from the IPMA onto the reference design. This was done by creating a C program that models the update and memory management algorithms that would be implemented in hardware. The results are presented in Table 3 .

Table 3: Analysis of Empirical Results

	Database Name				
	Mae East	Mae West	Pac-Bell	AADS	PAIX
Number of Prefixes	40902	19229	22246	23373	2994
Total Size	1.4 Mbits	702 Kbits	748 Kbits	828 Kbits	139 Kbits
Storage per Next Hop Pointer bit	2.85	3.04	2.8	2.95	3.87
Memory Mgmt.(MM) Overhead	20%	21%	13.5%	19.4%	21%
Total Size without MM	1.1 MBits	551 Kbits	636 Kbits	668 Kbits	109 Kbits
Storage per Next Hop Pointer bit (Without MM) {bits}	2.24	2.38	2.38	2.38	3.03

The first row in Table 3 repeats the prefix count for each database, the second row gives the total size of the table for each database. The third row gives the total number of bits stored for every bit of next hop

pointer. This number can be useful for comparing storage requirements with other lookup schemes since the size of the next hop pointer varies in the published literature for other schemes. The third row summarizes the percent overhead of the allocation headers used for memory management (MM). The next two rows of Table 3 are the table size and 'bits per next hop pointer bit' without the allocation headers counted. The reason the table size is recalculated without the memory management overhead, is that results for other IP address lookup schemes traditionally do not contain any memory management overhead.

The total storage per next hop pointer bit without memory management for Mae East is 2.24 bits per prefix. For Lulea[3] with the Mae-East database (dated January 1997 with 32k prefixes) 160 KB are required which indicates 39 bytes per prefix or 2.8 bits per next hop pointer bit (they use 14-bit next hop pointers). This analysis suggests that the reference design and of Tree Bitmap is similar in storage to Lulea but without requiring a complete table compression to achieve the results.

The reference design has assumed a range of possible main memory sizes from 16k nodes to 128k nodes. From the empirical evidence of 2.24 bits per prefix, it can be extrapolated that a 128k node memory system would be able store 143k prefixes. A 128k node memory system would require 4.8 mbits of SRAM, and ignoring routing channels and would occupy approximately 50% of a 12mmx12mm die in the IBM.25 micron process (SA-12). The logic to implement the reference lookup engine described here would require a very small amount of area compared to the SRAM.

8 Conclusions

We have described a new family of compressed trie data structures that have small memory requirements, fast lookup and update times, and is tunable over a wide range of architectures. While the earlier Lulea algorithm [3] is superficially similar to ours in that it uses multibit tries and bitmaps, our scheme is very different from Lulea. We use two bitmaps per trie node as opposed to one, we do not use leaf pushing which allows us to bound update times, we use a completely different encoding scheme for internal bit maps which requires a more involved search algorithm than finding the first bit set, and we push prefixes that are multiples of stride lengths to be associated with the next node in the path, even if it means creating a new node.

To do the complete processing in one memory access per trie node (as opposed to the three memory accesses required by Lulea), we keep the trie node sizes small by separating out the next hop information into a separate results node that is accessed lazily at the end of the search. We also described several further optimizations including split tree bitmaps and segmented bit maps to further reduce trie node size. None of these techniques appear in [3] or any earlier schemes.

We have also described a model for modern memory architectures that together with the family of Tree Bitmap variants allows us to pick the required algorithm for a given architecture. This model provides us with knowledge of the optimal burst size which in turn determines the stride size. Recall that for a particular set of optimizations, the stride size defines the bit map size and hence the burst size. The model also directly indicates the degree of pipelining required for maximum memory bandwidth utilization.

Using these trie algorithms and memory models, we described a competitive software algorithm and a hardware reference implementation. The hardware reference implementation presents a new memory management scheme (reactive compaction) that can be integrated with updates to give high memory utilization and easy table management (possibly in hardware). The reference implementation was shown to support over a hundred thousand prefixes with on-chip SRAM, support 25 million lookups per second (OC-192c), and guarantees of at least 10k updates per second. The memory management and updates can be hardware automated to result in an independent easy to manage lookup engine.

Together, our algorithms provide a complete suite of solutions for the IP lookup problem from the low end to the high end. They have the desired features of CAM solutions while offering a much higher

number of supported prefixes.

9 References

[1] Artisan Components, Inc., http://www.artisan.com/

[2] U. Black, S. Waters, *SONET & T1: Architectures for Digital Transport Networks*, Prentice Hall, 1997

[3] M. Degermark, A. Brodnik, S. Carlsson, S. Pink, “Small Forwarding Tables for Fast Routing Lookups” Proc. ACM SIGCOMM ‘97, , Cannes (14 - 18 September 1997).

[4] P. Gupta, S. Lin, N. McKeown, “Routing Lookups in Hardware at Memory Access Speeds,” Proc. IEEE Infocom ‘98.

[5] IBM, 64Mb Direct Rambus DRAM Advance Datasheet, December 1997.

[6] IBM, Double Data Rate Synchronous RAM, IBM0625404GT3B, September 1999

[7] IBM, Synchronous DRAM Datasheet, IBM031609CT3-322, 1- Meg x 8 x 2 banks, Janurary 1998

[8] IEEE Draft P802.3z/D3.0 “Media Access Control (MAC) Paramaters, Physical Layer, Repeater and Management Paramters for 1000Mb/s operation”, June 1997.

[9] S. Keshav, R. Sharma, “Issues and Trends in Router Design”, in IEEE Communications Magazine , May 1998.

[10] V.P. Kumar, T.V. Lakshman, D. Stiliadis, "Beyond Best-Effort: Gigabit Routers for Tomorrow's Internet," in IEEE Communications Magazine , May 1998.

[11] C. Labovitz, G. R. Malan, F. Jahanian. “Internet Routing Instability.” Proc. ACM SIGCOMM ‘97, p. 115-126, Cannes, France.

[12] B. Lampson, V. Srinivasan, G. Varghese, “IP Lookups using Multiway and Multicolumn Search,” Proc. IEEE Infocom ‘98.

[13] J. Manchester, J. Anderson, B. Doshi, S. Dravida, “IP over SONET”, IEEE Communications Magazine, May 1998.

[14] N. McKeown, “Fast Switched Backplane for a Gigabit Switched Router”, http://www.cisco.com/warp/public/733/12000/technical.shtml

[15] Merit Inc. IPMA Statistics. http://nic.merit.edu/ipma

[16] Micron, Syncburst SRAM Advance Datasheet, T58LC256K18G1, MT58LC128K32G1, MT58LC128K36G1, February 1998.

[17] Netlogic Microsystems, IPCAM-3 ternary CAMs. http://www.netlogicmicro.com/.

[18] S. Nilsson and G. Karlsson, "Fast address lookup for Internet routers",, Proceedings of IFIP International Conference of Broadband Communications (BC'98), Stuttgart, Germany, April 1998. To appear.

[19] C. Partridge et al., “A Fifty Gigabit Per Second IP Router”, IEEE Transactions on computing

[20] V. Srinivasan, G. Varghese, “Faster IP Lookups using Controlled Prefix Expansion,” Proc. SIGMETRICS ‘98.

[21] V. Srinivasan, G. Varghese, S. Suri, M. Waldvogel, “Fast Scalable Algorithms for Layer 4 Switching” Proc ACM SIGCOMM ‘98.

[22] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, “Scalable High Speed IP Routing Lookups,” *Proc. ACM SIGCOMM* ‘97, pp. 25-37, Cannes (14-18 September 1997).